

”„Echtzeit“ ist ein Buzzword und sagt gerade bei Nichtrealtime-Betriebssystemen  
garnichts aus.“  
(Thomas Ogrisegg in at.linux)

# Linux PREEMPT\_RT

Linux Realtime Patchset

Florian Westphal

15. November 2007

```
1024D/F260502D <fw@strlen.de>  
1C81 1AD5 EA8F 3047 7555  
E8EE 5E2F DA6C F260 502D
```

# Themenübersicht

- Einführung: Was ist 'Echtzeit'
- Latenz & Reaktionszeit
- Synchronisation/Locking
- Kernel Preemption
- -rt
- Besonderheiten von RT-Anwendungen

# Echtzeit

- Echtzeit  $\neq$  (besonders) schnell
- Echtzeit-System muss auf Ereignis innerhalb eines vorgegebenen Zeitrahmens reagieren
- garantierte (Reaktions-)Zeiten

→ Zuverlässigkeit, deterministisches Verhalten

- auch unter ungünstig und/oder unwahrscheinlichen Bedingungen
- Linux ist für "Common Case" optimiert

# Linux RT Ansätze

1. Linux Kernel als 'Prozess' unter einem Realtime-Betriebssystem
  - Kernel darf arbeiten, wenn keine RT-kritische Aktion ausgeführt wird
  - RT-kritische SW läuft außerhalb des Linux Kernels
2. Linux parallel zu RTOS auf eigener CPU
3. Linux -rt Patches (aka PREEMPT\_RT)

## Latenz und Reaktionszeit

Latenz: Zeitspanne zwischen Stimulus & Reaktion, z.B.

- Interrupt-Latenz: Zeit zwischen auftreten und Verarbeitung eines Interrupts
- Scheduling-Latenz: Zeitspanne zwischen aufwecken d. Prozesses und dem Zeitpunkt an dem der Prozess losläuft

Reaktionszeit wird auch durch

- Dauer von Interrupt-Handlern, Anzahl von Interrupts, etc.
- Zeit die das Scheduling selbst benötigt (Entscheidung, Kontextwechsel, etc.) beeinflusst.

## Ausflug: Linux & präemptives Multitasking

- Unix (und Linux) unterstützen Preemptives Multitasking
- . . . aber das bezieht sich 'nur' auf Prozesse, nicht auf Kernel-Code
- Linux 2.4: Nachdem ein Prozess einen Syscall ausführt, ändert sich der laufende Prozess nicht "unerwartet".  
Scheduling kann nur in 3 Fällen stattfinden:
  1. Programm wird im Userspace ausgeführt
  2. Programm wechselt von Kernel-Mode zurück in Userspace
  3. Kernel-Modus führt explizite Anweisung („schedule()“) aus (z.B. weil auf Hardware gewartet werden muss)
- Ausnahme: Interrupts unterbrechen laufenden Code

# Kerneltask-Synchronisation mit Interrupts

Code der Form "counter++" ist nicht *atomic*:

```
movl counter, %eax
incl %eax
movl %eax, counter
```

Beispiel: Senderoutine inkrementiert Sende-Warteschlangenzähler  
Interrupt Handler dekrementiert diesen.

## Beispiel

1. Senderoutine will inkrementieren: `movl counter, %eax` (läd Wert in Register, z.B. 1)
2. Interrupt trifft ein, Unterbrechung → Wert wird erneut in Register geladen, dekrementiert und gespeichert: jetzt 0
3. Nach Ende des Interrupt Handlers läuft Senderoutine wieder an – diese arbeitet mit dem Wert 1
4. "Wer zuletzt schreibt, schreibt am besten": Zähler steht nun auf 2



## (Historische) Lösung: Interrupts blockieren

```
local_irq_disable();  
counter++;  
local_irq_enable();
```

Bei SMP-Systemen reicht dies nicht aus:

- Der Interrupt-Handler kann von einer anderen CPU bearbeitet werden
- mehrere CPUs könnten die Senderoutine simultan ausführen

## Auf SMP Systemen ...

CPU 0

```
movl counter, %eax
```

```
incl %eax
```

```
movl %eax, counter
```

CPU1

```
movl counter, %eax
```

```
incl %eax
```

```
movl %eax, counter
```

→ Eine Addition geht verloren

→ Der Abschnitt darf nur jeweils von einer CPU ausgeführt werden („kritischer Abschnitt“)

# Spinlocks

- Synchronisationsmechanismus: Nur ein Task kann von `spin_lock(&lock)` "gesicherten" Abschnitt ausführen
- nutzt "atomare" CPU-Instruktion (z.B. `xchg`, lock prefix)
- Busy-Wait: Threads warten "aktiv" auf Lock-Freigabe → blockiert CPU
  - spinlocks nur für kurze Abschnitte verwenden
  - Thread, der spinlock hält, darf nicht "schlafen", also insb. keine Scheduler-Aufrufe
- Falls Synchronisation auch mit Interrupt-Kontext erforderlich: `spin_lock_irqsave()`
- Einprozessor-System: kein Spinlock notwendig

## Mutexe: Wenns mal länger dauert..

Mutexe sind spinlocks ähnlich. Wichtigste Unterschiede:

- Task darf (während Mutex gehalten wird) blockieren
- jeder Mutex verfügt über Warteliste
- Wenn Mutex belegt: Task an Warteliste anfügt und Scheduler-Aufruf → CPU nicht blockiert
- können nicht in Interruptkontext verwendet werden
- Zugriff auf Warteliste mit spinlock serialisiert

# Zusammenfassung

- Ablauf bestimmter Code-Sequenzen muss serialisiert werden
  - geteilte Datenstrukturen
  - Kommunikations mit Hardware/Geräten
- Interruptsperrern, spinlocks, mutexen
- Echtzeit  $\neq$  schnell
- Echtzeit-System muss auf Ereignis (Timer, externe Eingaben, etc.) innerhalb eines vorgegebenen Zeitrahmens reagieren

# Scheduling-Latenz: Ursachen

- Treiber mit aufwendigen/langen ISR
- Kernel-Code, welcher längere Zeitdauer in spinlock-Region verbringt
- → Scheduling kann währenddessen nicht durchgeführt werden
  
- Für schnelle Reaktionen/Antwortzeiten auf Ereignisse/externe Signale ist Rescheduling nötig (z.B. jetzt wieder lauffähiges Programm ausführen)

# Wie Scheduling-Latenz reduzieren?

- Scheduling nach auftreten eines Events (z.B. Interrupt) so schnell wie möglich durchführen
- Linux Kernel Preemption
  - Grundidee: Zeit zwischen Eintreten eines Ereignisses und Scheduler-Aufruf reduzieren
  - Mehr Gelegenheiten für Scheduler-Aufruf schaffen
  - Aber wann ist es sicher, den Kernel-Prozess zu unterbrechen?

kurze Antwort: Wenn der Kernel nicht einen kritischen Abschnitt abarbeitet.

# Explizites Scheduling

`cond_resched*()`-Funktionen

- Falls für längere Zeiträume keine Scheduling-Gelegenheit besteht
  - z.B., wenn über größere Datenmengen iteriert werden muss (`fs/`, `mm/. . .`)
  - → System weniger "Flüssig"
- Linux 2.4: "Lowlatency-Patches"
- Code-Anpassungen nötig ("lock breaking")



# CONFIG\_PREEMPT\_VOLUNTARY

- Mehr Codestellen an denen Scheduler-Aufrufe erfolgen
- Verwendet existierende Debugging-Funktionalität: `might_sleep()`
  - `might_sleep()` markiert Stellen, die zu scheduling-Aufrufen führen könnten
  - Mit aktiviertem `CONFIG_DEBUG_SPINLOCK_SLEEP` werden Stacktraces erzeugt, falls `might_sleep()` Aufruf innerhalb eines atomaren Kontextes (d.h. "hält spinlock" oder Interrupts abgeschaltet) erfolgt ist
  - Mit `CONFIG_PREEMPT_VOLUNTARY`:  
`might_sleep()` → `cond_resched()`

## include/linux/kernel.h

```
#ifndef CONFIG_PREEMPT_VOLUNTARY
extern int cond_resched(void);
# define might_resched() cond_resched()
#else
# define might_resched() do { } while (0)
#endif

#ifdef CONFIG_DEBUG_SPINLOCK_SLEEP
void __might_sleep(char *file, int line);
# define might_sleep() \
do{__might_sleep(__FILE__, __LINE__); might_resched();}while(0)
#else
# define might_sleep() do { might_resched(); } while (0)
#endif
```

# CONFIG\_PREEMPT

- aktueller Kernelthread kann verdrängt werden (nicht in spinlocks, Interrupts, etc.)
- spinlock-Aufruf inkrementiert/dekrementiert zusätzlich per-Prozess Zähler: `preempt_count`
- etwas vereinfacht:
  - nach Abarbeitung einer ISR sowie bei Aufruf von `spin_unlock(&x)`:
  - Wenn `preempt_count == 0` und Scheduler-Aufruf nötig (z.B. weil Task  $t$  nun lauffähig ist)  $\rightarrow$  `preempt_schedule()`

# Wie Reaktionszeit garantieren?

Was für RT kritische Bestandteile untersucht werden müsste:

1. Interrupthandler (incl. Softirqs)
2. Scheduler
3. Alle Stellen, ...
  - an denen Interrupts deaktiviert werden
  - an denen Preemption deaktiviert wird (incl. spinlocks)
4. Alle Code-Stellen an denen Mutexe/Semaphoren verwendet werden, die auch von den RT-Bestandteilen benötigt werden

# Priority Inversion-Problem (1)

Gegeben: Tasks  $t_1, t_2, t_3$ .

- $t_1$ : kritisch, RT Priorität
- $t_2$ : normale Priorität
- $t_3$ : niedrige Priorität
- $t_1$  und  $t_3$  teilen sich Mutex/Semaphore

## Priority Inversion (2)

1.  $t_3$ : holt mutex
2.  $t_1$ : wird ablauffähig, verdrängt  $t_3$
3.  $t_1$ : benötigt Mutex  $\rightarrow$  blockiert,  $t_3$  erhält CPU
4. diese "Priority Inversion" ist (noch) kein Problem
5. Wenn aber  $t_2$  lauffähig wird ist  $t_1$  auf *unbestimmte Zeit* blockiert (unbounded priority inversion)

## PREEMPT\_RT Patch: Ziele

- nicht-Unterbrechbare Codeanteile minimieren
- Anteil der Änderungen (gegenüber "normalem" Kernel) gering halten
- Mit PREEMPT\_RT werden (von wenigen Ausnahmen abgesehen)...
  - kritische Abschnitte
  - Interrupt-Handler
  - Abschnitte welche Interrupts sperren

. . . unterbrechbar
- Hochauflösende Timer
- Integration in Mainline

## ”Normaler Kernel”:

- drei Kontexte
  1. Interrupts (nicht unterbrechbar)
  2. Softinterrupts (nicht unterbrechbar)
  3. Prozess-Kontext (ausserhalb kritischer Abschnitte unterbrechbar)



# Kontexte mit PREEMPT\_RT

1. „Harte“ Interrupts (nicht unterbrechbar)
  - maskiert nur den jeweiligen Interrupt
2. Interrupt-Threads (unterbrechbar)
  - Je ein Thread per IRQ
  - Ruft alle "ISR" des jew. IRQ auf
3. Softinterrupts (unterbrechbar)
4. Prozess-Kontext (unterbrechbar)
  - Spinlock → RTMutex

# Priority Inheritance

Wdh: Priority Inversion

1.  $t_3$ : holt mutex
2.  $t_1$ : wird ablauffähig, verdrängt  $t_3$
3.  $t_1$ : benötigt Mutex  $\rightarrow$  blockiert,  $t_3$  erhält CPU

$\rightarrow$  Wenn nun  $t_2$  lauffähig wird, so ist  $t_1$  auf unbestimmte Zeit blockiert

Priority Inheritance "vererbt" Priorität von  $t_1$  an  $t_3$

$\rightarrow$  stellt sicher, dass Prozess nie von einem (niedriger priorisierten) Prozess blockiert wird

# RT-Mutexes

- Mutexe mit PI-Protokoll
- Ursprünglich -rt, inzwischen in Mainline  
(für `pthread_mutex_t` `PTHREAD_PRIO_INHERIT`)
- Eigentümer eines RT-Mutexes erhält Priorität von wartenden Tasks mit höherer Priorität
- Falls der Eigentümer selbst auf einen anderen RT-Mutex blockt, Priorität ggf. neu vererben

## RT-Mutexes (forts.)

- RT-Mutex-Warteliste: nach Priorität sortiert
- `pi_list`: Per-Task Liste mit 'Top-Waiters' aller RTMutexe des Tasks, nach Priorität sortiert
- Wenn sich erster Task dieser Liste ändert: Priorität d. Eigentümers anpassen

## Reader/Writer Locks (rwlock\_t)

- es wird zwischen "lesenden" und "schreibenden" Nutzern unterschieden
  - rwlock kann von *mehreren* lesenden *oder einem* schreibenden Task verwendet werden
  - Wenn schreibender Zugriff benötigt: Es muss auf *alle* Leser gewartet werden
  - Problem: Blockiert auf Unbestimmte Zeit
1. Only one task at a time may read-acquire a given reader-writer lock.
  2. If #1 results in performance or scalability problems, the problematic lock will be replaced with RCU (read-copy update).

## RCU (read-copy update)

- Schreiber blockieren keine Leser
- Leser halten Schreiber nicht davon ab, Daten zu verändern

→ keine Priority Inversion möglich

## RCU (read-copy update)

- Löschooperationen werden zweigeteilt
  1. verhindern, dass zu löschende Daten noch sichtbar sind
  2. Löschooperation durchführen
- RCU ermöglicht lockfreie Lesezugriffe auf Daten, z.B. Linked List
  1. Element aus der Liste herauslösen
  2. Element löschen, nachdem kein Leser mehr eine Referenz auf dieses Element hält

## RCU (forts.)

- normaler Kernel: Preemption wird während read-Zugriff abgeschaltet (`rcu_read_lock()`)
  1. Leser dürfen nicht schlafen; (RCU kritischer Abschnitt); nicht bei `PREEMPT_RT`
  2. Schreiber haben mehr arbeit: alte Daten, auf die noch zugegriffen werden kann müssen vorgehalten werden

Quiz: Wie erkennt der RCU-Mechanismus, wann alte Daten freigegeben werden können?



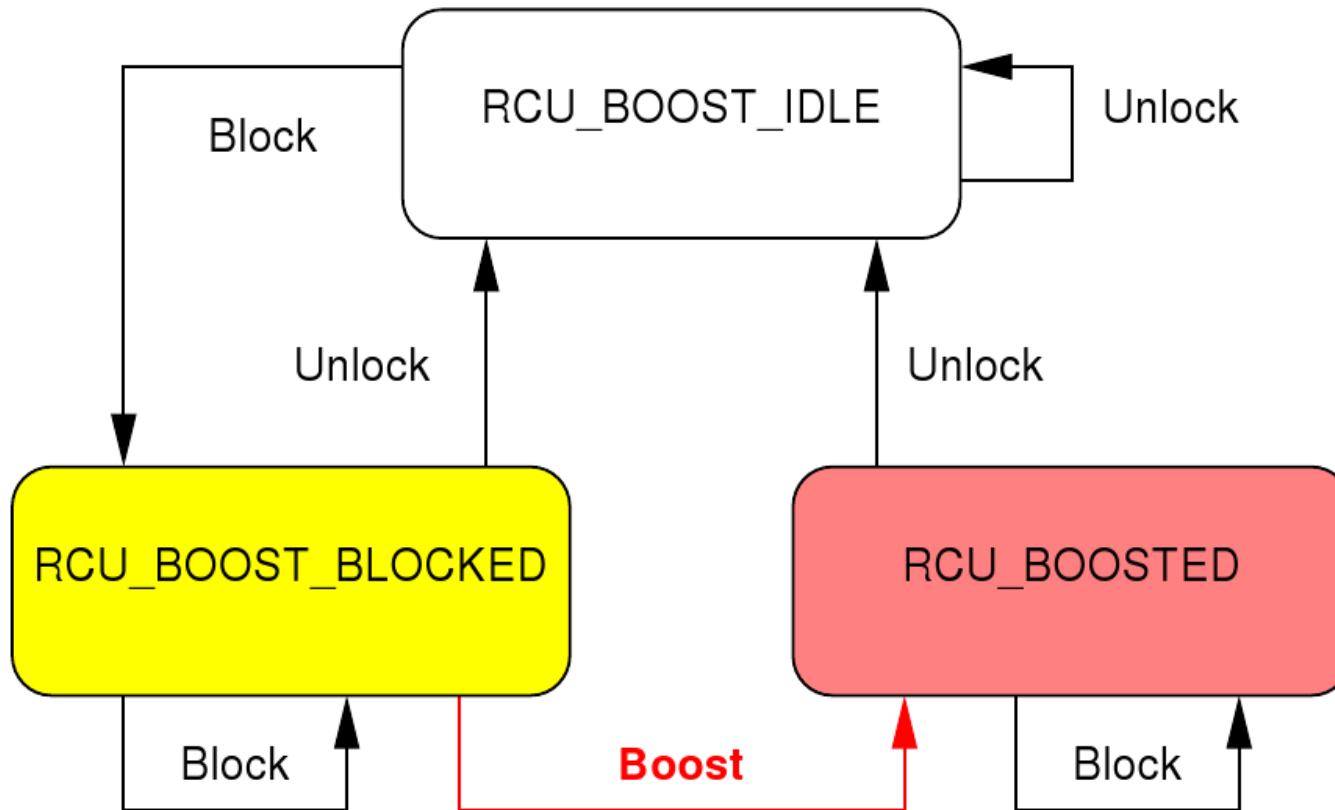
## RCU im -rt Patchset

- RCU-Leseseite wird preemptiv: `current->rcu_read_lock_nesting`
- Neues Problem:
  - Wenn (viele) Leseseiteteilen zu lange blockieren: OOM
  - Gesucht: "was ähnliches wie Priority Inheritance"

## RCU Priority Boosting

- RCU-Booster Task: überwacht (per-CPU) Liste mit Tasks welche RCU-Leseseitig blockiert sind
- Overhead nur dann, wenn Task blockiert ist; keine Änderung an `rcu_read_lock` etc.
- Scheduler prüft `current->rcu_read_lock_nesting`, wenn  $> 0$ : Pre-emption innerhalb RCU-Readseite
  - Task in Priority-Boost-Liste eintragen
- Task trägt sich selbst aus Liste aus wenn äußerste (blockierte) Leseseite beendet ist
- Booster Task kontrolliert Liste periodisch

# Zustände



(Quelle: [2])

# Linux Scheduling Klassen

- **SCHED\_OTHER**: Prioritäten (-20 (hoch) bis 19 (niedrig))
  - Die Standard-Klasse. Alle Prozesse erhalten faire CPU-Anteile. Höher Priorisierte Prozesse erhalten längere/häufigere Zeitscheiben, blockieren niedrigere Prozesse jedoch nicht.
- **SCHED\_FIFO**: Echtzeitpriorität. Es wird jeweils der höchstpriorisierte Prozess bearbeitet. Solange der Prozess nicht blockiert, wird die CPU nicht abgegeben ("for (;;) ;" mit SCHED\_FIFO → alle anderen Prozesse laufen nicht mehr)
- **SCHED\_RR**: 'SCHED\_FIFO mit Zeitscheiben'; bei mehreren SCHED\_RR Prozessen gleicher Priorität laufen diese abwechselnd

Einstellen mit: `chrt(1)`, `sched_setscheduler(2)`

## RTOS allein reicht nicht aus

- Hardware-Latenzen: SMI/SMM, DMA, . . .
  - SM-Interrupts liegen außerhalb der Kontrolle des Betriebssystems
  - nicht vorhersagbare/kontrollierbare Unterbrechungen
- Pagefault-Latenzen:
  - Verwendung von Dateien
  - Swap
  - mmap, fopen, . . .

## RTOS allein reicht nicht aus

Anwendung muss auch geeignet entworfen werden, z.B.

- Teilen von Anwendung in RT-kritische und weniger wichtigen Teil
- Auslagern von (bestimmten) Speicherseiten mit `mlock(2)`/bzw. `mlockall(2)` deaktivieren
- allen Speicher vor betreten des RT-Bereichs anfordern *und* initialisieren

# FIN

- mit RT-Patch wird Kernel fast vollständig preemptible
- Dazu werden z.B. ISR etc. in Kernel-Threads "ausgelagert"  
→ deterministischeres Zeitverhalten auf Kosten der Leistung
- Highres-Timer, Userspace Priority Inheritance in Mainline

# Literatur

- 1 Darren Hart, Steve Rostedt, "Internals of the RT Patch"; Linux Symposium 2007.
- 2 Paul McKenney "Priority-Boosting RCU Read-Side Critical Sections"  
<http://www.rdrop.com/users/paulmck/RCU/RCUbooststate.2007.04.16a.pdf>  
(RCU-Seite: <http://www.rdrop.com/users/paulmck/RCU/>)
- 3 RT-Wiki: <http://rt.wiki.kernel.org/>
- 4 RT-Patches: <http://www.kernel.org/pub/linux/kernel/projects/rt/>
- 5 [Documentation/rt-mutex-design.txt](#)